Symulacja i testowanie mikroprocesora MicroBlaze na płycie XSB

Autor: Zespół Rekonfigurowalnych Systemów Obliczeniowych www.fpga.agh.edu.pl
Ostatnia aktualizacja: 22.03.2005

1 Wstęp

Zadaniem tego dokumentu jest pokazanie jak zaprojektować i zasymulować moduł składający się z mikroprocesora MicroBlaze. Projekt składa się z następujących modułów:

- mikroprocesor MicroBlaze
- pamięć zewnętrzna *opb_sram*
- moduł pamięci wewnętrznej BRAM na magistrali OPB i LMB, moduły interface'u pomiędzy tymi magistralami oraz pamięcią BRAM *opb_bram_if_cntrl, lmb_bram_if_cntrl*; w pamięci BRAM znajduje się pamięć programu procesora MicroBlaze.
- moduł *opb_epp* – komunikacja pomiędzy komputerem PC a płytą XSB poprzez port równoległy pracujący w trybie EPP oraz środowisko APSI służące do komunikacji z płytą XSB z poziomu komputera PC
- moduł *opb_uartlite* – UART do komunikacji z PC poprzez port szeregowy np. w celu testowania procesora MicroBlaze.
- moduł *util_clkdiv* moduł służący do zmniejszenia częstotliwości zegara taktującego cały system do 50MHz/parametr. Opóźnienia w układzie są często większe niż dopuszczalne dla 50 MHz dlatego, oryginalna częstotliwość 50MHz jest zmieszona.

Zadaniem tego projektu będzie:

- Zapoznanie się z pakietem EDK (Embedded Development Kit) firmy Xilinx i soft-procesorem MicroBlaze
- Zapoznać się z elementem bibliotecznym opb_epp, opb_sram i środowiskiem APSI
- Zapoznać się z metodami symulacji układu na poziomie języka VHDL, syntezy oraz symulacji czasowej
- Zapoznać się z metodami zadawania wektorów wymuszeń dla skomplikowanych układów takich jak moduł opb_epp oraz pamięci zewnętrznej SRAM
- Przetestowanie układu w rzeczywistych warunkach za pomocą wewnętrznego analizatora stanów logicznych.
- Debugowanie procesora MicroBlaze w układzie FPGA poprzez port UART.

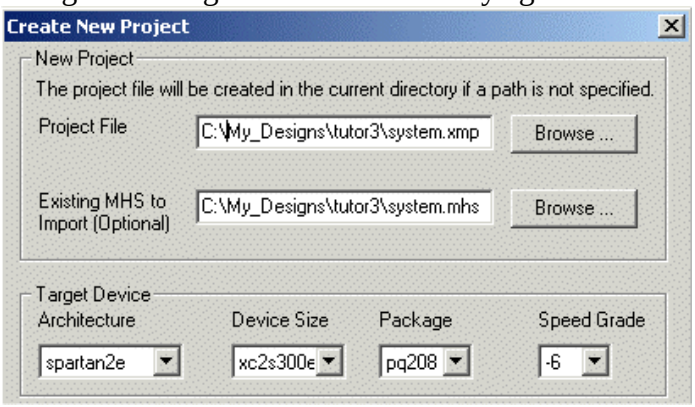
2 Projekt w EDK

Przed rozpoczęcie projektu w EDK należy wykonać następujące czynności:

- W katalogu c:/*my*_designs stworzyć nowy katalog (w Windows Commander lub Explorer) o wybranej przez siebie nazwie (w naszym wypadku tutor3). Katalog ten od tej pory będzie nazywany katalogiem roboczym EDK. Następnie stworzyć podkatalog

o nazwie pcores (w naszym wypadku c:/my_designes/tutor3/pcores). Uwaga: ścieżka dostępu nie może zawierać spacji czyli ścieżka *C:/moje dokumenty/tutor3* jest niepoprawna.

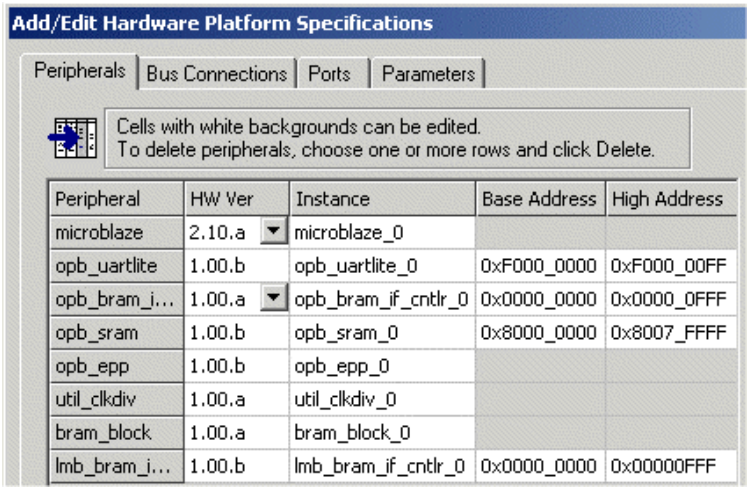
- Ze strony *http://galaxy.uci.agh.edu.pl/~jamro/opb* skopiować (i rozpakować) do katalogu pcores moduł: *apsi*, *opb_epp*, *opb_sram*, *util_clkdiv* oraz dane do tego tutorialu (dane do tego tutorialu znajdują się zaraz po opisie tego tutorialu w akapicie Tutoriale). Ściągnięte pliki należy rozpakować do katalogu: katalog_roboczy_edk\pcores. (W ten sposób powstanie np. katalog: katalog_roboczy_edk\pcores\opb_sram_v..., itd.). Przy pobieraniu modułów *opb_epp* oraz *opb_sram* należy zwrócić uwagę, że dotyczą one płyty XSB a nie płyty XSV.
- Otworzyć program EDK: Programy/Xilinx Embeded Development Kit/Xilinx Platform Studio
  - Stworzyć nowy projekt przez wybranie menu: File/New Project o parametrach przedstawionych na poniższym rysunku. Ścieżka *Project File* powinna być taka sama jak katalog roboczy EDK. Aby przyspieszyć wykonywanie projektu poprzez pominięcie następnego rozdziału należy skopiować plik *system.mhs* (z danych do tego tutorialu) do katalogu roboczego EDK oraz zaznaczyć go w ścieżce *Existing MHS*.



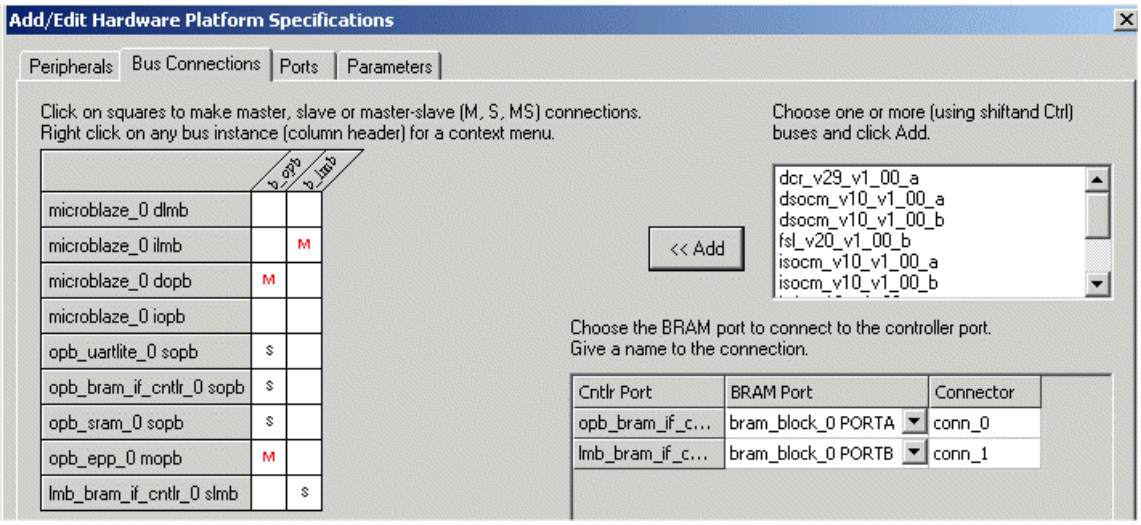
3 Określenie Schematu w EDK

W tym rozdziale zostanie określony schemat połączeń różnych urządzeń czyli plik *.mhs. Jeżeli w poprzednim punkcje został określony plik *.mhs to należy tylko zapoznać się z tym rozdziałem nie wykonując żadnej czynności. Rozszerzenie MHS pochodzi od Microprocessor Hardware Specification.

Pierwszym etapem projektu jest osadzenie modułów w projekcie. W tym celu należy wybrać menu: Project/Add Edit Cores. Następnie dodaj elementy biblioteczne zgodnie z podanymi poniżej rysunkami:

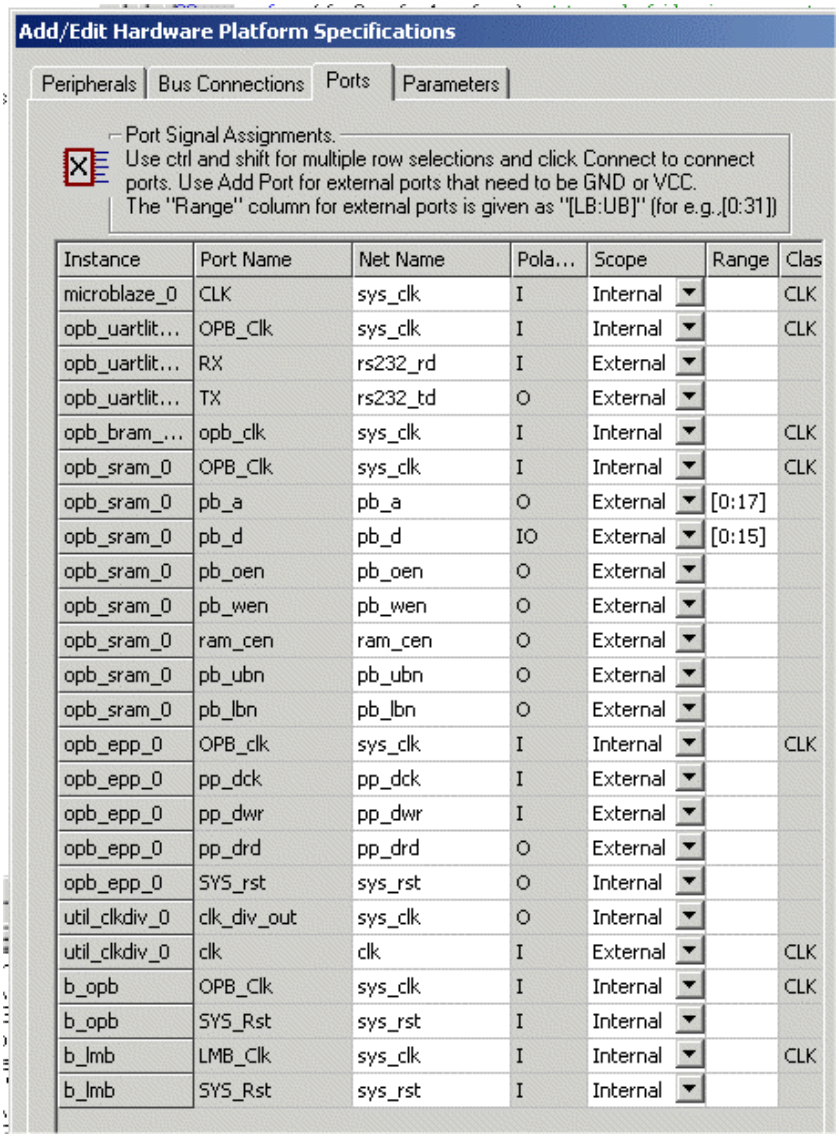


Peripheral	HW Ver	Instance	Base Address	High Address
microblaze	2.10.a	microblaze_0		
opb_uartlite	1.00.b	opb_uartlite_0	0xF000_0000	0xF000_00FF
opb_bram_i...	1.00.a	opb_bram_if_cntlr_0	0x0000_0000	0x0000_0FFF
opb_sram	1.00.b	opb_sram_0	0x8000_0000	0x8007_FFFF
opb_epp	1.00.b	opb_epp_0		
util_clkdiv	1.00.a	util_clkdiv_0		
bram_block	1.00.a	bram_block_0		
lmb_bram_i...	1.00.b	lmb_bram_if_cntlr_0	0x0000_0000	0x00000FFF



Warto w tym miejscu podkreślić że procesor MicroBlaze posiada dwie niezależne magistrale dla danych (dopb) i programu (ilmb). W naszym projekcie magistrale te mają dostęp do tej samej pamięci BRAM (pamięć BRAM jest dwuportowa i każda z magistral ma dostęp do tej pamięci poprzez inny port). Aby dodać magistralę do projektu należy zaznaczyć magistrale w naszym przypadku OPB (wersja dowolna) lub LMB oraz nacisnąć przycisk ADD. Można zmienić nazwę magistrali na *b_opb* i *b_lmb*.

Następnie należy wybrać zakładkę ports, która opisuje wszystkie zewnętrzne wyprowadzenia. Opisywane są też połączenia wewnętrzne, które są poza standardem magistrali OPB – sygnały magistrali OPB są łączone automatycznie. Przykładem sygnałów wewnętrznych (ang. internal) jest sygnał *sys_rst* – resetujący cały system generowany przez moduł opb_epp na podstawie odpowiedniego zapisu przez port równoległy. Aby dodać sygnał wybrany w prawym oknie, należy przycisnąć przycisk ADD. W przypadku dodawania wielu sygnałów zalecane jest używanie klawisz Ctrl. Aby połączyć sygnały już dodane, należy je zaznaczyć w lewej części okna i nacisnąć klawisz Connect. Zwróć uwagę czy jest to sygnał wewnętrzny czy zewnętrzny oraz czy jest to magistrala czy też pojedynczy sygnał (kolumna range).



Instance	Port Name	Net Name	Pola...	Scope	Range	Clas
microblaze_0	CLK	sys_clk	I	Internal		CLK
opb_uartlit...	OPB_Clk	sys_clk	I	Internal		CLK
opb_uartlit...	RX	rs232_rd	I	External		
opb_uartlit...	TX	rs232_td	O	External		
opb_bram_...	opb_clk	sys_clk	I	Internal		CLK
opb_sram_0	OPB_Clk	sys_clk	I	Internal		CLK
opb_sram_0	pb_a	pb_a	O	External	[0:17]	
opb_sram_0	pb_d	pb_d	IO	External	[0:15]	
opb_sram_0	pb_oen	pb_oen	O	External		
opb_sram_0	pb_wen	pb_wen	O	External		
opb_sram_0	ram_cen	ram_cen	O	External		
opb_sram_0	pb_ubn	pb_ubn	O	External		
opb_sram_0	pb_lbn	pb_lbn	O	External		
opb_epp_0	OPB_clk	sys_clk	I	Internal		CLK
opb_epp_0	pp_dck	pp_dck	I	External		
opb_epp_0	pp_dwr	pp_dwr	I	External		
opb_epp_0	pp_drd	pp_drd	O	External		
opb_epp_0	SYS_rst	sys_rst	O	Internal		
util_clkdiv_0	clk_div_out	sys_clk	O	Internal		
util_clkdiv_0	clk	clk	I	External		CLK
b_opb	OPB_Clk	sys_clk	I	Internal		CLK
b_opb	SYS_Rst	sys_rst	I	Internal		
b_lmb	LMB_Clk	sys_clk	I	Internal		CLK
b_lmb	SYS_Rst	sys_rst	I	Internal		

Dokonaj edycji zakładki PORTS aby otrzymać dane zgodne z poniższym rysunkiem, należy połączyć:

- Wszystkie sygnały *OPB_clk*, *LMB_clk* oraz sygnał *clk_div_out* w jeden sygnał wewnętrzny *sys_clk*.
- Zewnętrzny sygnał zegara *clk* podłącz do wejścia *clk* modułu *util_clkdiv*.
- Wszystkie sygnały *opb_epp* zaczynające się na *pp_** należy dodać i poddać edycji usuwając przedrostek *opb_epp_0*; Uwaga dla płyty XSV należy dodatkowo podać zakres (range) dla sygnału **pp_d[0:7]**.
- dla sygnałów UART'a wyprowadź *rs232_td* oraz *rs232_rd*; Uwaga: Dla płyty XSV należy użyć nazw*: uart_rx* i *uart_tx*.
- dodaj i połącz razem sygnały *sys_rst* modułu *b_opb* (magistrala OPB), *b_lmb* (magistrala LMB) i *opb_epp* (moduł generujący sygnał reset poprzez odpowiedni zapis przez port równoległy) – sygnał wewnętrzny;
- dodaj wszystkie sygnały modułu *opb_sram*, usuń przedrostek *opb_sram_0_*; zwróć uwagę na zakres (range): pb_d[0:15] i pb_a[0:17]. Uwaga: Dla płyty XSV należy zmienić nazwy sygnałów na ladr[0:18], ldata[0:15], loen, lcen, lwen.

Następnym etapem jest wybranie zakładki parameters, gdzie zostaną określone parametry wybranych modułów (określonych słowem *generic* w języku VHDL). Opis wybranego

modułu oraz jego parametrów można uzyskać wybierając dany moduł oraz przyciskając przycisk Open PDF Doc.

W zakładce Parameters należy ustawić następujące parametry dla poszczególnych modułów:
opb_epp
- c_la_mwidth= 8 – liczba próbek 2^8 (256) na sygnał dla analizatora stanów logicznych
- c_la_dwidth=64 – liczba oglądanych sygnałów w analizatorze stanów logicznych
- c_la_twidth= 64 – liczba sygnałów wejściowych analizowanych przez układu triggera
- c_la_cewidth= 64 – liczba sygnałów wejściowych analizowanych przez układu zezwolenia zegara

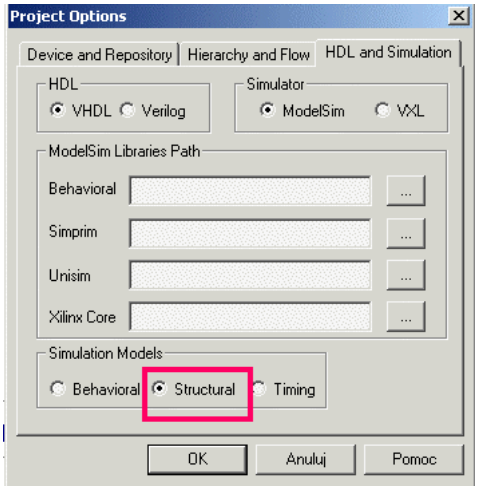
b_opb (magistrala OPB lokalna procesora MicroBlaze)
- c_dynam_priority=1 – priorytet przydzielania magistrali OPB jest dynamiczny

opb_uartlite
- c_clk_freq= 12_500_000 – częstotliwość zegara systemowego (12.5MHz)
- c_baund_rate= 9600 – prędkość transmisji.

util_clkdiv
c_div_factor=1 – dzielenie zegara 50MHz przez *c_div_factor*. W niektórych syyuacjach opóźnienia wewnątrz układu FPGA są zbyt duże, w takim wypadku należy zmniejszyć częstotliwość zegara poprzez odpowiednie ustawienie tego parametru.

4 Synteza oraz generacja modeli do symulacji

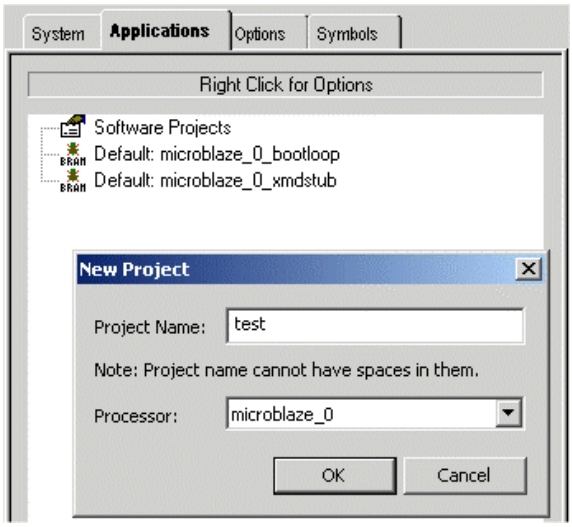
Następnie należy wybrać opcje projektu (zobacz poniższy rysunek): Option/Project Option/HDL and Simulation/Simulation Models/Structural.



Po wykonaniu powyższych czynności należy wygnerować netlistę (zsyntezować) opisanego projektu, menu: Tools/Generate Netlist oraz wygenerować model do symulacji menu: Tools/Sim Model Generation.

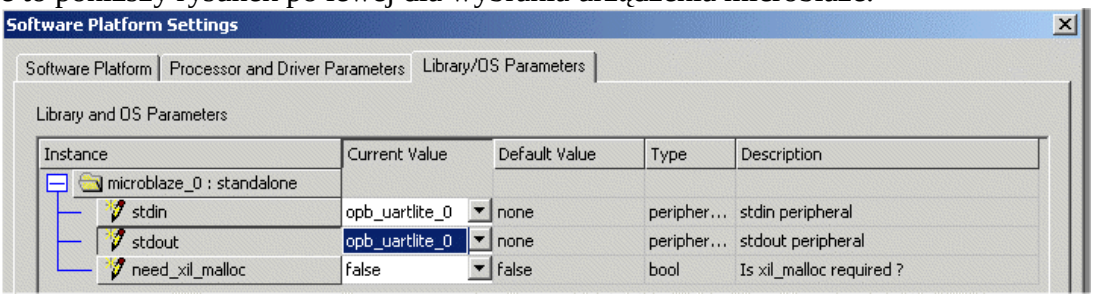
5 Generacja kodu programu mikroprocesora w EDK

Pierwszą czynnością jest utworzenie nowego projektu software'owego. W tym celu należy wybrać zakładkę Applications oraz przycisnąć prawym przyciskiem myszki napis: *Software Projects* i wybrać opcje: *Add New Project*. Pokazuje to poniższy rysunek:



Następnie kiedy został stworzony nowy projekt (o nazwie *test*) Przyciskamy prawym przyciskiem myszki nazwę tego projektu. W ten sposób możemy określić parametry projektu poprzez opcje *Set Compiler Options*. W zakładce *Optimizations* wybieramy opcje: *Create symbols for debugging.* W ten sposób możliwe będzie śledzenie kodu wygenerowanego przez kompilator.

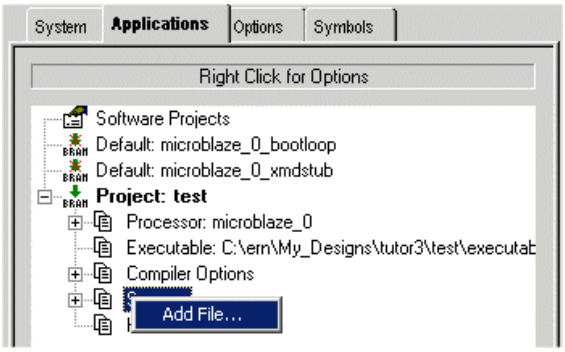
Dla każdego urządzenia można określić poziom sterowników poprzez zaznaczenie danego urządzenia oraz naciśnięcie prawego klawisza myszki i wybranie S/W Settings. Pokazuje to poniższy rysunek po lewej dla wybrania urządzenia microblaze.



Warto podkreślić, że dla procesora można zaznaczyć urządzenie STDIN, STDOUT czyli domyślne urządzenie wejścia wyjścia na którym będą np. wyprowadzane komunikaty (może to być np. UART). Również możliwe jest ustawienie różnego poziomu driver'a, czy też system operacyjny.

Następną czynnością jest generacja bibliotek na podstawie podanego hardware'u, wybierz menu: Tools/Generate Libraries. Podczas tej czynności generowany jest np. plik *xparameters.h* wykorzystywany poniżej.

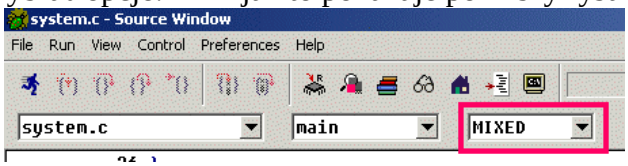
Następnie skopiuj do katalogu roboczego EDK plik *system.c* z katalogu dane do tutorialu. Następnie w pakiecie EDK określ ten plik jako źródło programu (*Sources*) dla procesora microblaze jak to pokazuje poniższy rysunek:



Naciśnij dwukrotnie ten plik aby go oglądnąć i ewentualnie poddać edycji.
Zwróć uwagę na linię *#include "xparameters.h"*. Plik *xparameters.h* opisuje parametry całego systemu np. adres bazowy urządzenia. Zapoznaj się z dalszą częścią programu.

Następnie należy skompilować kod poprzez wybranie komendy menu: *tools/build all user applications.* W ten sposób wygenerowany zostanie plik test/executable.elf.

Symulacja na poziomie języka C
Wybierz menu: tools/software debugger. Następnie aby zobaczyć kod wynikowy należy w software debuggerze wybrać opcje: MIX jak to pokazuje poniższy rysunek:



Możliwa jest częściowa symulacja software'owa kodu procesora MicroBlaze w tym programie. Ma ona jednak wiele ograniczeń, przede wszystkim nie jest możliwe symulacja części hardware'owej czyli wartości pamięci zewnętrznej, rejestrów urządzeń peryferyjnych, itd. Prawidłowo symulowana jest tylko pamięć podłączona do portu LMB. Dlatego symulacja ta jest zalecana tylko na poziomie podstawowych instrukcji, które i tak można sprawdzić w standardowym symulatorze języka C. Możliwe jest używając tego programu połączenie się z płytą XSV i debuggowanie procesora.

6 Symulacja VHDL

W tym punkcie symulowane zostanie całe środowisko sprzętowe w języku VHDL. Możliwe jest również symulowanie wykonywania kodu programu, jednakże jest to bardzo utrudnione, ponieważ dostępna jest tylko wersja binarna instrukcji. PO drugie symulacja jest relatywnie wolna.

Otworzyć program ActiveHDL i stworzyć nowy projekt. Parametry programu są dowolne (nie trzeba wybierać narzędzia do syntezy i implementacji, nieważny jest też wybrany układ FPGA). W dalszej części tego tutorialu będziemy się odwoływać do katalogu *katalog_AHDL/nazwa_projektu/src* jako do katalogu roboczego ActiveHDL.

Akapit tylko do przeczytania: Istnieją dwa rodzaje symulacji VHDL:

1) po syntezie, kiedy to opis modułu z języka VHDL został zamieniony na niższy poziom rejestrów bramek, itd. Na tym poziomie praktycznie nie ma możliwości symulować strukturę wewnętrzną danego modułu. Symulacja po syntezie umożliwia jednak wykrycie ewentualnych błędów wynikłych ze złego opisu języka danego modułu (np. napisany kod VHDL nie jest dobrze syntezowalny) lub też ze źle działającego narzędzia syntezującego. Aby przesymulować dany moduł na poziomie po syntezie należy go skopiować z katalogu

katal_roboczy_EDK/simulation/structural/nazwa_modułu_wrapper.vhd do katalogu roboczego AHDL.

2) Aby symulować moduł funkcjonalnie – czyli tak jak on został napisany w języku VHDL należy skopiować następujące pliki: a) opis parametrów (*generic)* zewnętrznych danego modułu czyli plik *katalog_roboczy_edk\hdl\nazwa_modulu_wrapper.vhd* oraz opis samego modułu czyli pliki: *katalog_roboczy_edk\pcores\nazwa_modulu\hdl\vhdl*.vhd* do katalogu roboczego ActiveHDL. Jeżeli dany moduł jest elementem własnym (nie dostarczonym przez firmę Xilinx, np. opb_sram) to jego opis znajduje się bezpośrednio w katalogu pcores (jak to opisano powyżej). Jeżeli element jest elementem dostarczonym przez firmę Xilinx to zamiast katalogu katalog_roboczy_EDK\pcores należy użyć katalogu *EDK_program\hw\iplib\pcores*. Niestety niektóre moduły dostarczone przez firmę Xilinx (np. microblaze) są zakodowane i można z nich korzystać tylko na poziomie po syntezie.

Ogólna zasada jest taka, że w katalogu *simulation* znajdują się kompletne pliki po syntezie a w katalogu hdl znajdują się pliki vhdl, które wymagają dołączenia dodatkowych plików źródłowych znajdujących się np. w katalogu pcores.

W naszym projekcie będziemy korzystać z modułu MicroBlaze'a, uartlite, opb_bram_if_ctrl oraz magistrali OPB tylko na poziomie po syntezie (nie są dostępne na poziomie HDL) oraz z pozostałych modułów na poziomie języka HDL.

W dalszej części tego ćwiczenia należy wykonać jedną z dwóch poniższych czynności:
a) Skopiować pliki z katalogu *vhdl_sim* (dane do tutorialu: plik *tut_log_anal.zip*) do katalogu roboczego ActiveHDL. W przypadku wybrania tego punktu należy przejść do punktu Wymuszenia.

b) Skopiować następujące pliki z katalogu roboczego EDK do katalogu roboczego ActiveHDL:
hdl\opb_epp_0_wrapper.vhd – plik opisujący parametry układu opb_epp na poziomie języka HDL
hdl\opb_sram_0_wrapper.vhd – plik opisujący parametry układu opb_sram na poziomie języka HDL
hdl\util_clkdiv_0_wrapper.vhd – plik opisujący parametry układu opb2opb_dwidth na poziomie języka HDL
simulation\b_opb_wrapper.vhd – opisujący działanie magistrali oraz arbitra OPB (po syntezie)
simulation\b_lmb_wrapper.vhd – opisujący działanie magistrali LMB (po syntezie)
simulation\bram_wrapper.vhd – opisujący działanie modułu BRAM (po syntezie)
simulation\microblaze_0_wrapper.vhd – opisujący działanie MicroBlaza (po syntezie)
simulation\opb_uartlite _0_wrapper.vhd – opisujący działanie układu UART (po syntezie)
simulation\opb_bram_if_ctrl_0_wrapper.vhd – opisujący działanie interface'u pomiędzy magistralą OPB i pamięcią BRAM (po syntezie).
simulation\lmb_bram_if_ctrl_0_wrapper.vhd – opisujący działanie interface'u pomiędzy magistralą LMB i pamięcią BRAM (po syntezie).
simulation\system.vhd – plik nadrzędny opisujący cały system (cały układ FPGA), plik dostępny tylko na poziomie HDL.
simulation\system_init.vhd – plik umożliwiający inicjalizacje pamięci programu BRAM dla celów symulacyjnych. Podczas symulacji procesor MB może pobierać poprawny kod programu.

pcores\opb_epp\others\epp_model.vhd – plik symulujący działanie portu równoległego EPP na podstawie pliku apsi.txt czytanego przez program apsi.exe
pcores\opb_epp\others\la_view.vhd – plik umożliwiający wizualizacje sygnałów zarejestrowanych przez analizator stanów logicznych
pcores\opb_epp\hdl\vhdl*.vhd – plik opisujący moduł opb_epp na poziomie VHDL
pcores\opb_sram\hdl\vhdl*.vhd – plik opisujący moduł opb_sram na poziomie VHDL
pcores\util_clkdiv\hdl\vhdl*.vhd – plik opisujący moduł util_clkdiv na poziomie VHDL
pcores\opb_sram\others\sram_model.vhd – model symulacyjny zewnętrznej pamięci SRAM. Model ten czyta na starcie zawartość pliku sramin.bin i iniclalizuje tym plikiem pamięć zewnętrzną SRAM.
pcores\opb_sram\others\sramin.bin – przykładowy plik binarny, którym podczas symulacji inicjalizowana jest pamięć zewnętrzna SRAM. W przypadku kiedy pamięć programu jest zamieszczona w pamięci zewnętrznej
pcores\apsi.exe – program wykonawczy do komunikacji pomiędzy komputerem PC a płytą XSB.
tutorial\test.vhdl – plik nadrzędny do symulacji symulujący środowisko zewnętrzne pliku system.vhd (wymuszający sygnał clk, sygnały portu równoległego EPP, pamięci zewnętrznej SRAM. Katalog *tutorial* to dane (zip) do tego tutiarialu.
tutorial\apsi_*.txt – skrypt (komendy) wykonywane przez program apsi.exe
tutorial\test.hex – przykładowy plik z danymi

Skopiowane pliki *.vhd należy dodać do projektu ActievHDL a następnie skompilować.
Uwaga: Podczas kompilacji modułów VHDL należy ręcznie kasować następujące niedostępne biblioteki pokazywane przez ActiveHDL:
LIBRARY opb_epp_v1_00_a;
USE opb_epp_v1_00_a.ALL;
Jeżeli po podwójnej kompilacji w AHDL wszystkich plików przy jakimś pliku pojawi się wykrzyknik to należy poprawić błędy, np. nie został skopiowany i dodany do projektu AHDL jakiś plik.

Wymuszenia

VHDL

Naturalną i najczęściej stosowaną (przez studentów) metodą symulacji jest podawanie wymuszeń podczas symulacji w programie ActiveHDL. Jest to jednakże metoda bardzo prymitywna i dlatego stosowana tylko dla bardzo prostych projektów.

Następną bardziej rozbudowaną metodą podawania wymuszeń częściowo stosowaną podczas tego projektu jest nadawanie wymuszeń na poziomie języka VHDL. W tym celu napisano specjalny moduł test.vhd, w którym osadzono moduł testowany: *system.vhd* (moduł nadrzędny). Otwórz plik test.vhd i zwróć uwagę na następujące fragment modułu:

```
process begin
wait for 10 ns;
clk<= '0';
wait for 10 ns;
clk<= '1';
end process;
```

Powyższy fragment powoduje generacje sygnału zegarowego. Przykład jest bardzo prosty (można go również zaimplementować na poziomie symulacji waveform ActiveHDL) ale może być w prosty sposób rozbudowany do bardzo skomplikowanego, niemożliwego do zrealizowania w waveform'ie. Bardziej zaawansowany przykład, który nie może być łatwo

zastosowany na poziomie symulacji waveform, jest symulacja działania układu UART, w naszym przykładzie bezpośrednie połączenie sygnałów rs232_td z rs232_rd, przez co to co zostało wysłane może być bezpośrednio odebrane podczas symulacji.

sram_model.vhd

Przykładem bardziej zaawansowanej symulacji jest symulacja pracy pamięci zewnętrznej SRAM. Pamięć SRAM może być zapisywana i odczytywana wiele tysięcy razy pod różne lokacje adresowe. Dlatego ręczna symulacja takiej pamięci jest praktycznie niemożliwa. Dlatego został napisany specjalny moduł *sram_model.vhd*, który emuluje pamięć zewnętrzną SRAM. Jednym z ograniczeń takiego modelu pamięci SRAM jest pojemność pamięci SRAM. Symulacja dużej pamięci SRAM jest bardzo czasochłonna i zasobożerna. Dla przykładu, żeby zapamiętać jeden bit pamięci SRAM nie wystarczy jeden bit pamięci operacyjnej komputera na którym odbywa się symulacja. Jest tak ponieważ każdy bit w pamięci SRAM podczas symulacji może być reprezentowany za pomocą różnych dodatkowych symboli dostępnych w standardzie std_logic: '0', '1', 'X', 'U', itd. Ponadto symulator często musi pamiętać zawartość całej pamięci SRAM w różnych chwilach czasowych, co dodatkowo zwiększa zapotrzebowanie na zasoby komputera symulującego. Dlatego zalecane jest symulowanie tylko części niezbędnej pamięci SRAM. W pliku *test.vhd* dodatkowa stała *sram_awidth* określa szerokość symulowanej magistrali adresowej a przez to i wielkość symulowanej pamięci SRAM.

Pamięć SRAM podczas symulacji jest na inicjalizowana z pliku (nazwa domyślna sramin.bin) a po zakończeniu symulacji zawartość pamięci SRAM jest zapisywana do pliku (sramout.bin).

Pewną niedogodnością modułu sram_model jest to, że **rozmiar pliku sramin.bin musi być większy od rozmiaru pamięci** w przeciwnym przypadku **generowany jest błąd**. Dlatego w przypadku gdy rozmiar pliku jest mniejszy niż rozmiar pamięci zaleca się dołączenie do pliku sramin.bin dodatkowych dowolnych danych. Można to zrobić np. w programie apsi.exe skryptem *append.txt*, który powoduje podwojenie długości pliku *sramin.bin*. W tym celu uruchom: *apsi.exe append.txt*. Warto podkreślić że rozmiar pamięci jest określony za pomocą adr_width oraz data_width. W przypadku pamięci 16-bitowej (data_width=16) należy rozmiar pamięci pomnożyć przez 2. Można również dołączyć dodatkowe dane do pliku sramin.bin w innym programie np. Windows Commander. W projekcie założono rozmiar pamięci 2kB.

Inicjalizacja pamięci BRAM programu

W przypadku kiedy pamięć programu znajduje się w pamięci zewnętrznej SRAM, inicjalizacja pamięci odbywała się poprzez odpowiedni odczyt pliku *sramin.bin* (zob. wcześniejszy podrozdział). W innym przypadku kiedy pamięć programu znajduje się w pamięci wewnętrznej BRAM, inicjalizacja pamięci BRAM odbywa się w inny sposób. EDK podczas generacji plików do symulacji generuje w katalogu *simulation* specjalny plik o nazwie *system_init.vhd*, w którym znajdują się wartości inicjujące pamięć programu BRAM. Aby jednak zainicjować pamięć programu w pliku nadrzędnym do symulacji (*test.vhd)* została użyta specjalna składnia konfiguracji podana poniżej:

```
configuration TESTBENCH_FOR_system of test is
for test_arch
for UUT : system
use configuration work.system_conf;
end for;
end for;
```

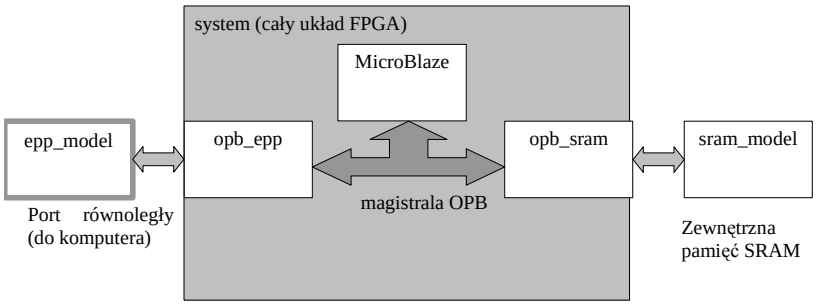
end TESTBENCH_FOR_system;

W konsekwencji jako moduł nadrzędny podczas symulacji należy zaznaczyć ***configuration TESTBENCH_FOR_system***.

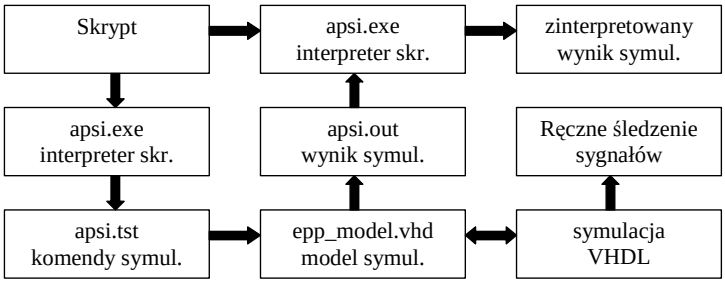
W przypadku braku wcześniej wspomnianej procedury pamięć BRAM podczas symulacji zostanie zainicjalizowana samymi zerami. W niektórych przypadkach możliwe jest inicjalizowanie pamięci BRAM poprzez środowisko *APSI (*zob. rozdział Inicjalizacja pamięci programu), nie jest to jednak zalecane. Warto zwrócić uwagę, że zmienienie kodu programu wymaga również odświeżenie pliku *system_init.vhd* poprzez wybranie w EDK menu: *Tools/Sim model Generation* oraz skopiowanie wspomnianego pliku do katalogu roboczego AHDL.

epp_model (teoria)

Przykładem bardziej zaawansowanej symulacji jest plik *epp_model.vhd*, który współpracuje z programem *apsi.exe* i umożliwia symulowanie działania portu równoległego EPP oraz działanie programu *apsi.exe*. Moduł ten należy umieścić w pliku nadrzędnym do symulacji (*test.vhd*) w miejscu gdzie fizycznie znajduje się port równoległy pokazuje to poniższy rysunek.



Dzięki temu rzeczywiste działania wykonywane przez program apsi.exe mogą być symulowane w ActiveHDL bez potrzeby czasochłonnego pisania osobnych skomplikowanych wymuszeń działania portu równoległego na poziomie języka VHDL. Schemat blokowy całego procesu symulacji jest przedstawiony poniżej:



Poszczególne elementy powyższego rysunku są następujące:

1) Skrypt

Pierwszą czynnością jest określenie komend jakie ma wykonywać program APSI, czyli napisanie odpowiedniego skryptu. Jedyną czynnością dodatkową podczas symulacji w porównaniu ze standardowym wykonywaniem jest dodanie na początku skryptu dodatkowej instrukcji *vhdlsim.*

2) Uruchomienie programu *apsi.exe*

Ten krok służy generacji wymuszeń dla symulatora. Ponieważ w skrypcie umieszczono instrukcje *vhdlsim*, program *apsi.exe* przestanie się komunikować z płytą XSV a zacznie generować wymuszenia dla symulatora w postaci pliku o roboczej nazwie *apsi.tst*.

3) Komedy symulacyjne *apsi.tst*
Pliku *apsi.tst* służący do komunikacji pomiędzy interpreterem skryptu (*apsi.exe)* a symulatorem VHDL.
4) Model symulacyjny portu równoległego *epp_model.vhd*
Model ten należy osadzić w pliku symulacyjnym (testbench) w miejscu gdzie fizycznie znajduje się port równoległy.
5) Symulacja VHDL
Podczas symulacji możliwe jest śledzenie przebiegów sygnałów wewnętrznych i zewnętrznych w podobny sposób jak to się odbywa podczas standardowej symulacji.
6) Wynik symulacji *apsi.out*
Wynik symulacji można analizować bezpośrednio podczas symulacji, jednakże bardzo często liczba przebiegów praktycznie uniemożliwia sprawdzenie poprawności działania układu. Dlatego dodatkowo podczas symulacji wszystkie dane odczytane na porcie równoległym są zapisywane przez model symulacyjny *epp_model.vhd* do pliku *apsi.out.*
6) Powtórne uruchomienie programu *apsi.exe*
Podczas powtórnego uruchomienia interpreter dysponuje wynikiem symulacji zapisanym w pliku *apsi.out* i na podstawie tego pliku zachowuje się tak jakby dane zapisane w tym pliku zostały w rzeczywistości odczytane z portu równoległego. Podsumowując, podczas powtórnego uruchomienia skrypt zachowuje się tak jakby w rzeczywistości kontaktował się z płytą XSV podłączoną do portu równoległego.

Przykładowy skrypt do symulacji

Aby zasymulować działanie naszego układu należy wykonać pewne czynności w systemie, które zostały opisane w skrypcie *apsi_sram_transf.txt* wykonywanym przez w program *apsi.exe*. Dokładny opis instrukcji skryptu znajduje się w pliku apsi.doc w projekcie APSI.

Zawartość pliku *apsi_sram_transf.txt*

vhdlsim // program apsi.exe przechodzi w tryb symulacji
config system // configuracja ukladu FPGA (instrukcja nie symulowana)

baseadr 8000_0000 // adres bazowy pamięci sram
writebyte 123 55 // zapisz pojedynczy bajt
readbyte 123 // odczytaj pojedynczy bajt

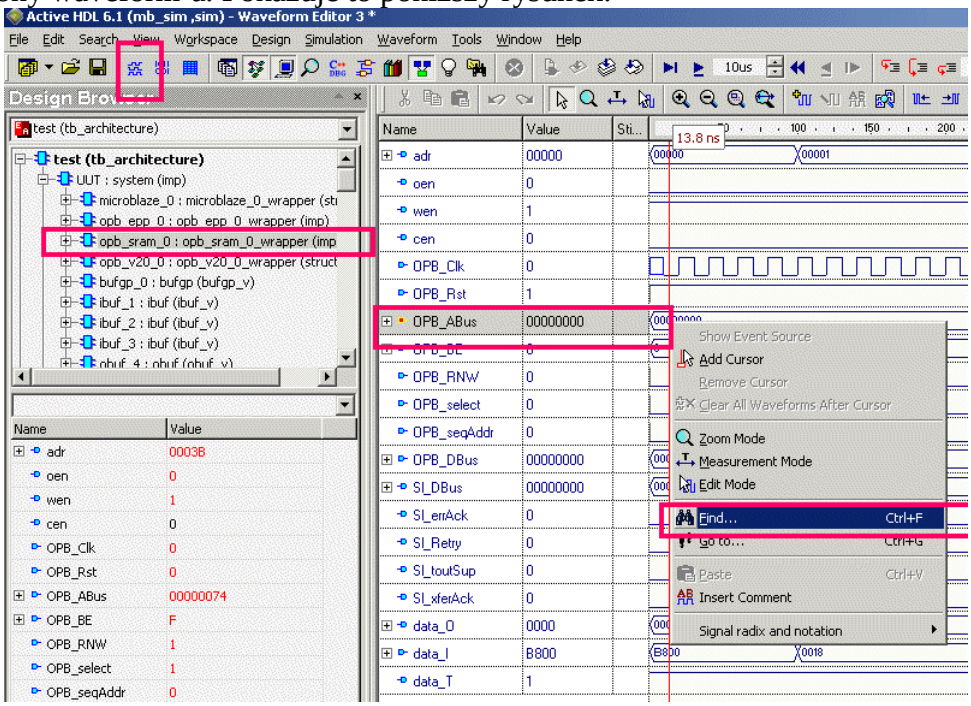
//transfer danych do pamieci SRAM
writeblock test.hex 120 12F // zapisz pamiec programu
// odczytaj to co zostalo ostatnio zapisane
readblock test.tmp 120 12F
// sprawdz poprawnosc zapianych danych – porównaj pliki
filecomp test.hex test.tmp
waitforkey // czekaj aż zostanie naciśnięty jakiś klawisz

Aby zasymulować działanie powyższego skryptu uruchom program: *apsi.exe apsi_sram_transf.txt.* Pliki *apsi.exe* i *apsi_sram_transf.txt* powinny być skopiowany do katalogu roboczego ActiveHDL'a. Podczas pierwszego uruchomienia programu wynik odczytu jest inny od faktycznego ponieważ nie przeprowadzono jeszcze symulacji.

Symulacja VHDL właściwa

Następnym etapem jest przeprowadzenie symulacji projektu w programie ActiveHDL.

Pierwszą rzeczą jest wybranie elementu nadrzędnego, którym jest element *test **(configuration TESTBENCH_FOR_system)***. Następnie należy zainicjalizować symulacje, menu: Simulation/Initialize Simulation. Teraz należy wybrać sygnały, które chcemy oglądnąć poprzez stworzenie nowego waveform'a (4 ikona od lewej, zob. poniższy rysunek) wybranie modułu opb_sram_wrapper oraz przeciągnięcie go (trzymając lewy przycisk myszki) na nowo stworzony waveform'a. Pokazuje to poniższy rysunek:



W ten sposób możemy oglądać wszystkie sygnały które znajdują się w bloku *opb_sram_wrapper.vhd*. Następnie uruchom symulacje. Jeżeli wybierzemy bardzo duży czas symulacji to symulacja jest automatycznie kończona w momencie kiedy zostaną wykonane wszystkie komendy skryptu. Jest to oznajmione poprzez komunikat w ActiveHDL'u*: FAILURE: O.K. End of simulation (by epp_model, no further commands to execute).*

Podstawowe sygnały na magistrali OPB:
OPB_ABus – adres
OPB_DBus – dana
OPB_select – urządzenie master transferuje dane (wystawia ważny adres)
OPB_RNW – urządzenie master określa kierunek transmisji (Read /Not Write)
OPB_xferAck – urządzenie slave jest gotowe do przesłania danych

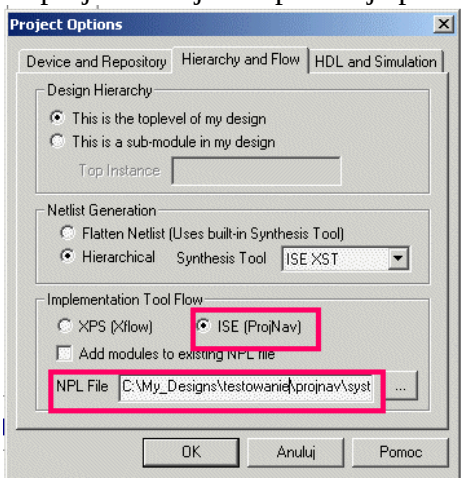
Spróbuj prześledzić kiedy wykonywany jest zapis i odczyt z pamięci SRAM, pod jaki adres i jakie dane.

Teoretycznie na podstawie oglądanych przebiegów symulacyjnych możliwe jest sprawdzenie czy dane zapisywane i odczytywane z pamięci są poprawne. Jednakże jest to dość czasochłonne i możliwa jest tylko wybiórcza kontrola. Dlatego aby przekonać się czy dane odczytane są poprawne należy powtórnie uruchomić program: *apsi.exe apsi_sram_transf.txt*. Tym razem program ten dysponuje już danymi odczytanymi z portu równoległego podczas symulacji i zapisuje faktyczne dane do pliku binarnego *test.tmp*. Przy bardzo długich plikach ręczne przeglądanie odczytanych danych nie jest jednak praktyczne dlatego program *apsi.exe* ma komendę porównywania dwóch plików *filecomp*.

7 Implementacja

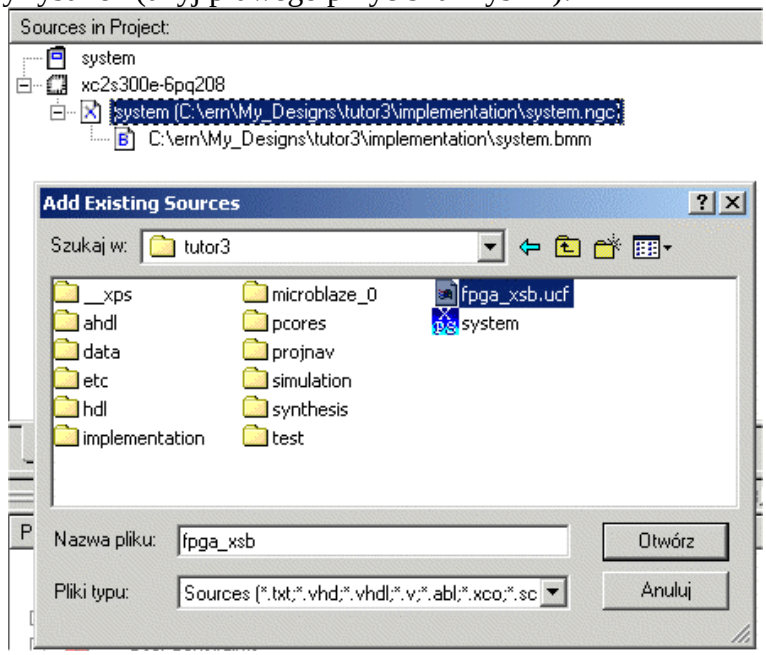
W wyniku implementacji zostaje wygenerowany plik system.bit programujący układ FPGA. W niektórych przypadkach (brak czasu na zajęciach Testowanie ...) można pominąć wykonywanie tego rozdziału poprzez użycie gotowego pliku system.cfg dostępnego w pliku dane do tutorialu.

Wrócić do programu EDK i wyeksportuj projekt do programu Xilinx'a Project Navigator poprzez wybranie menu: Tools/Export to ProjNav, aby to było możliwe należy wybrać opcje w menu: Option/Project Option projekt ISE jak to pokazuje poniższy rysunek:

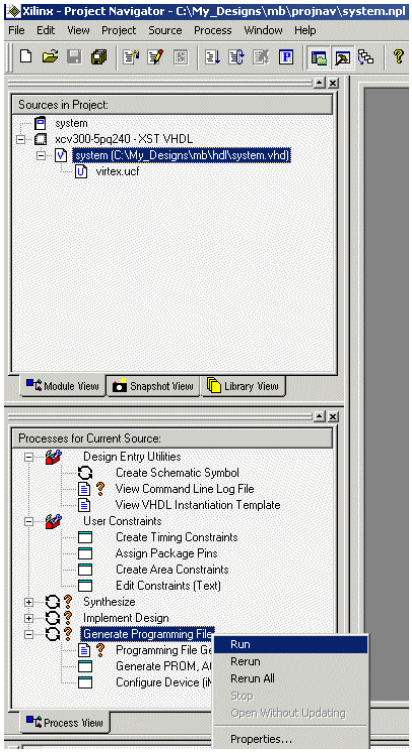


Ścieżka robocza projektu programu Project Navigator zostanie ustawiona w opcjach Hierarchy and Flow/NPL File (zob. powyższy rysunek) i do tego katalogu zostanie wyeksportowany projekt z EDK.

Następnie należy otworzyć program Project Navigator oraz projekt który przez chwilą został wygenerowany przez EDK. Do tego projektu należy dołączyć plik *fpga_xsb.ucf* (*virtex.ucf* dla płyty XSV), który między innymi opisuje lokalizacje końcówek układu FPGA. Plik *fpga_xsb.ucf* należy skopiować z katalogu tutorial do katalogu roboczego Project Navigator. Plik *fpga_xsb.ucf* (skopiuj z katalogu tutorial) należy dołączyć do projektu jak to pokazuje poniższy rysunek (użyj prawego przycisku myszki):



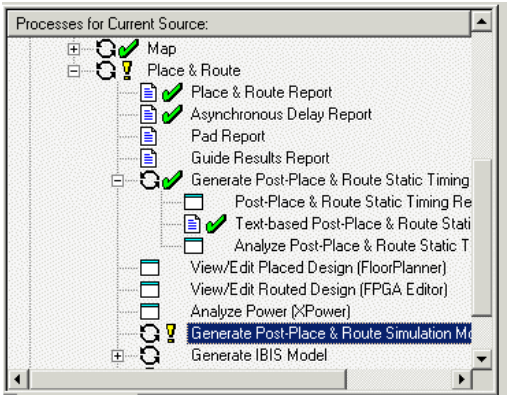
Następnie uruchom program generujący bit'a (plik konfigurujący układ FPGA), jak to pokazuje poniższy rysunek:



8 Symulacja czasowa

Punk ten należy wykonać tylko w przypadku dużej ilości wolnego czasu.

Aby przeprowadzić symulacje czasową (uwzględniającą rzeczywiste opóźnienia czasowe w układzie FPGA) należy w programie Project Navigator wybrać Gennerate Post Place & Route jak to pokazuje poniższy rysunek:



W ten sposób został wygenerowany plik system_timesim.vhd w katalogu roboczym Project Navigator. Następnie skopiuj ten plik do katalogu roboczego ActiveHDL. Plik ten powinien podmienić istniejący opis vhdl pliku system.vhd. Można tego dokonać np. przez utworzenie nowego projektu (design i skopiowanie do niego wszystkich plików oprócz system.vhd) lub skasowanie biblioteki znajdującej się w okienku Design Browser w zakładce Files oraz powtórną kompilacje następujących plików: epp_model.vhd, system_timesim.vhd, test.vhd. Symulacja czasowa przebiega w podobny sposób jak dla pozostałych opisanych uprzednio symulacji, jednakże decydującą różnicą jest to, że sygnały wewnętrzne są zakodowane. Dlatego podczas tej symulacji najlepiej jest oglądać tylko sygnały zewnętrzne układu virtex. Sygnały wewnętrzne są dostępne w sposób trudny do rozszyfrowania, z praktycznego punktu

widzenia możliwe jest tylko oglądanie sygnałów będących wyjściem rejestrów. Dlatego w sytuacjach gdy symulacja czasowa nie przebiega poprawnie (wynik jest niepoprawny) bardzo często sygnały które chcemy oglądnąć należy po prostu wyprowadzić na zewnątrz.

Symulacja czasowa jest czasochłonna i często nie pokazuje dokładnie problemu w opóźnieniach czasowych dlatego zaleca się użycie programu ISE Post-Place & Route Static Timing Analyzer, który analizuje opóźnienia czasowe w układzie i pokazuje najgorsze opóźnienia. Program ten jest dostępny w Project Navigator w oknie Processes and Current Sources / Place & Route / Generate Post Place Route and Static .. / Post-Place and Route Static Timing …

9 Inicjalizacja pamięci programu

W wyniku implementacji został wygenerowany plik *system.bit*, którym można konfigurować układ FPGA. Niestety ten plik nie zawiera poprawnej zawartości pamięci programu - pamięć BRAM jest inicjalizowana samymi zerami. Zainicjalizować pamięć BRAM można na 2 sposoby: poprzez środowisko EDK lub poprzez środowisko APSI.

Sposób 1

Jest to sposób preferowany, kiedy pamięć programu jest zawarta w pamięci wewnętrznej BRAM. Po wygenerowaniu pliku konfigurującego układ FPGA należy wykonać następujące czynności: W menu programu EDK wybierz: Tools/Import form ProjNav. Następnie zaznacz następujące pliki: *projnav/system.bit* oraz *implementation/system_bd.bmm*. Powyższa czynność ustawia, który plik konfigurujący (*.bit) ma być zmieniony oraz jakie jest ułożenie pamięci blokowych BRAM w danym pliku konfiguracyjnym.

Następnie w menu EDK wybierz *Tools/Update Bitstream.* Czynność ta powoduje, że zawartość pamięci BRAM w pliku konfigurującym układ FPGA zostanie zmieniona zgodnie z kodem programu wykonywanym przez procesor MB. Wyniku powyższej czynności zostanie wygenerowany plik *download.bit*. Warto w tym miejscu podkreślić, że jakakolwiek zmiana kodu programu nie wymaga teraz powtórnej generacji pliku konfigurującego w programie ProjNav. Konieczna jest tylko powtórna kompilacja programu oraz wybranie menu *Tools/Update Bitstream.* Ta czynność spowoduje uaktualnienie zawartości pamięci BRAM i zapisanie wyniku w pliku *download.bit.*

Sposób 2

W tej opcji kod programu zostaje zapisany poprzez środowisko APSI. Sposób ten należy zastosować przede wszystkim kiedy kod programu znajduje się w pamięci zewnętrznej. Może on być również stosowany w przypadku użycia wewnętrznej pamięci programu ale jest mniej zalecany.

Pierwszą czynnością jest wygenerowanie kodu binarnego z pliku *executable.elf*, który powstał w wyniku kompilacji kodu *C*. W menu EDK wybieramy: *tools/xygwin shell*. W uruchomionym programie obowiązują komendy unix'a. Należy zmienić katalog na katalog gdzie znajduje się plik *executable.elf* komendami: *cd test*. Następnie należy uruchomić komendy:

mb-objcopy executable.elf sramin.bin –O binary* konwersja pliku executable.elf na plik binarny *sramin.bin* reprezentujący dokładnie zawartość pamięci. Uwaga: * Parametr *–O* to litera O jak Ola.

mb-objdump -d executable.elf > objdump.txt disasemblacja kodu programu

Oglądnij powstały plik objdump.txt. Plik sramin.bin zawiera kod programu, który będzie wykonywany.

Następną czynnością jest załadowanie pamięci zewnętrznej (wewnętrznej) plikiem *sramin.bin* można tego dokonać używając komend środowiska APSI:
writeblock sramin.bin 0 FFFF zapisz plik sramin.bin do pamięci pod adres 0 do FFFF (lub mniejszy)
writebyte FFFF_FFFF FF // ustaw reset całego sytemu na 1
writebyte FFFF_FFFF 00 // ustaw reset na 0

10 Rzeczywista weryfikacja

Plik system.bit / *download.bit* (lub plik system.cfg / *download.cfg*, skopiowany z danych do tego tutoriala) wygenerowany przez Project Navigator należy teraz skopiować do katalogu roboczego ActiveHDL. Następnie otwórz plik *apsi_mb.txt*. W pliku tym usunięto linię vhdlsim – co spowoduje że program apsi.exe przejdzie w tryb rzeczywistej pracy i komunikacji z płytą XSV. Aby jednak układ działał poprawnie należy najpierw przesłać kod programu do pamięci SRAM oraz zresetować system.
Następnie uruchom program *apsi.exe apsi_mB.txt*. Program automatycznie odczyta dane zapisane przez procesor MB i sprawdzi czy są one poprawne.

11 Analizator stanów logicznych

Analizator został opisany w pliku pcores\opb_epp\doc\log_anal.doc oraz pcores\opb_epp\doc\opb_epp.doc

Aby analizator był użyty w parametrach modułu opb_epp należy ustawić c_la_mwidth>4.
W tym projekcie założono następujące parametry modułu opb_epp czyli:
c_la_mwidth= 8, c_la_dwidth= 64, c_la_twidh= 64, c_la_cewidth= 64

Aby oglądnąć rzeczywiste przebiegi rejestrowane w analizatorze stanów logicznych należy uruchomić skrypt *apsi_la.txt*:

```
// Rejestracja pracy MicroBlaza za pomoca analizatora
config download // configuracja ukladu FPGA (instrukcja nie symulowana)

// ustawienia analizatora stanow logicznych
la_baseadr FFFF_0000 // adres bazowy analizatora
latvalue 000000_0000_0000_04 //
latcare  000000_0000_0000_04 // analizator wyzwalany resetem systemu
lastart 1 // uaktywnin analizator (wyzwolenie na poczatku)

writebyte 3F0 01 // zapisz semafor

// reset sytemu po transferze danych
baseadr FFFF_FFFF // adres bazowy modułu reset
writebyte 0 1 // ustaw reset 1
writebyte 0 0 // ustaw reset 0

//transfer danych do pamieci SRAM
baseadr 8000_0000 // adres bazowy pamięci SRAM
writeblock sramin.bin 0 F // zapisz pamiec
readblock sramin.tmp 0 F
filecomp sramin.bin sramin.tmp // sprawdz poprawnosc zapianych danych

// czekaj az program zostanie zakonczony
waitbit0 FF 3F0 1000// koniec programu (komorka semaphore= 0)

readblock wynik.bin 400 040F // czytaj zawartosc zmodyfikowanej pamieci
filecomp wynik.bin wzor.hex // porownaj wynik (symulacji) z wzorem (poprawnym wynikiem)
```

laread "la_data.bin" 100
waitforkey

W wyniku tego uruchomienia zostanie wygenerowany *plik la_data.bin*, w którym zawarte są zarejestrowane przez analizator dane. Następnie należy skopiować do katalogu roboczego ActiveHDL plik *opb_epp\others\la_view.vhd*. Plik ten należy wybrać jako nadrzędny (top level). Plik *la_view.vhd* służy do oglądania w ActiveHDL danych zarejestrowanych przez analizator stanów logicznych (czyli danych zapisanych w pliku *la_data.bin*).

Następnie należy zainicjalizować symulacje, dodać nowy waveform oraz dodać wszystkie sygnały modułu la_view. Następnie należy uruchomić symulacje z dużym czasem symulacji ponieważ symulacja i tak zostanie zakończona w momencie kiedy zostaną wyświetlone wszystkie dane zarejestrowane przez analizator (komunikat: *FAILURE: O.K. All acquired data in the LA has already been shown*). Wyświetlone dane odpowiadają danym zarejestrowanym przez analizator.

Niestety transfer danych przez port równoległy EPP jest stosunkowo wolny w porównaniu z zegarem systemowym 50MHz, dlatego rejestrowane dane są najpierw zapisywane w pamięci wewnętrznej analizatora a następnie transmitowane do komputra off-line. Wielkość wewnętrznej pamięci jest jednak ograniczona a przez to ograniczona jest również liczba oglądanych przebiegów.

Dla parametru c_la_dwidth= 64 oraz c_la_dtype= 0 oglądane są następujące dane:

```
OPB_select<= d(0);
OPB_xferAck<= d(1);
OPB_rst<= d(2);
OPB_timeout<= d(3);
OPB_errAck<= d(4);
OPB_RNW<= d(5);
OPB_seqAddr<= d(6);
OPB_busLock<= d(7);
OPB_ABus<= d(31 downto 8);
OPB_DBus<= d(63 downto 32);
```

Trigger jest taki sam jak oglądane dane (parametr c_la_dtype= 0). Aby wyzwolić analizator w momencie kiedy procesor ma dostęp do danych zawartych pod adresem 1FF0 należy użyć komendy skryptu apsi.exe.

la_tvalue 0001FF0_0000_0000_00 // ustaw trigger na adres 1FF0
la_tcare FFFFFF_0000_0000_00 // podczas wyzwalania brany jest pod uwagę tylko stan magistrali adresowej

Dodatkowo moment wyzwolenia jest pokazywany w pliku la_view poprzez sygnał trigger. Wartość trigger'a (wartości danych, które wyzwalają analizator) jest pokazywana za pomocą sygnału trigger_value. Sygnały *la_mem* oraz *d* należy ignorować – wewnętrzne sygnały analizatora.

Aby rejestrować tylko stany przesłań na magistrali OPB należy włączyć dodatkowy obwód zezwolenia zegara komendami (należy wyjąć je z komentaża)

lacevalue 2
lacecare 2
lacectr 4 // logika ce jest uzywana do wybiorczego rejestrowania sygnalow

Aby oglądać tylko chwile kiedy procesor kontaktuje się z UART'em oraz chwile aktywne (Ack='1') należy wybrać instrukcje:

lacevalue F0_0000_0000_0000_02 // UART adres i ack=1 adres UARTu= 00F0_0000
lacecare FF_FFF0_0000_0000_02

lacectr 4 // logika ce jest uzywana do wybiorczego rejestrowania sygnalow

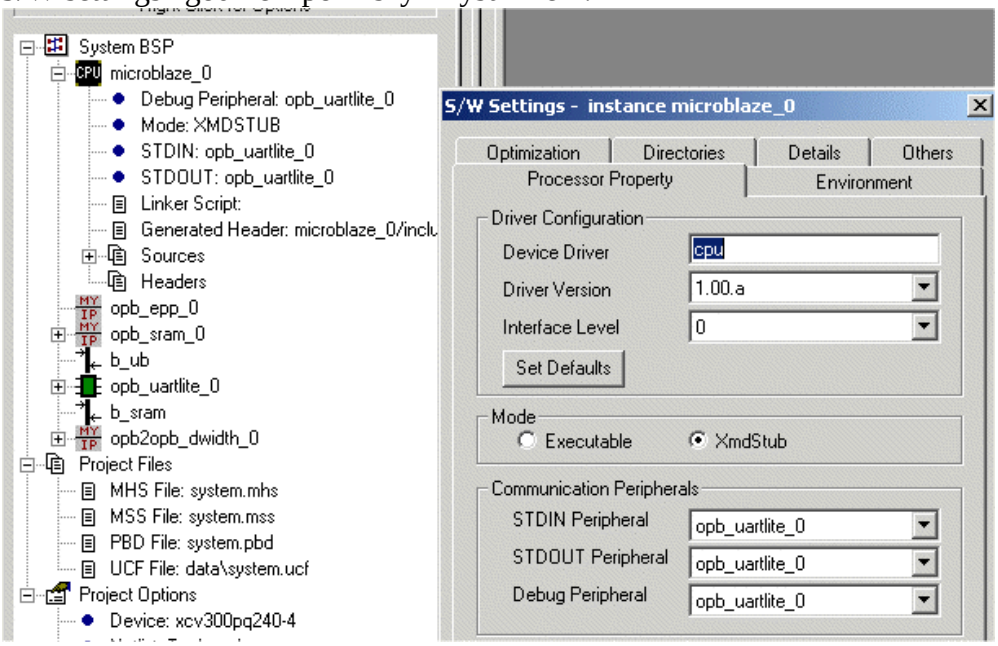
Zadania dodatkowe

- Jaki jest czas dostępu do pamięci SRAM (dla zapisu i odczytu).
- Spróbuj zakończyć rejestracje w momencie kiedy następuje zakończenie wykonywania programu (odpowiednie ustawienie triggera oraz komemda *lastart 3F)*
- Spróbuj zarejestrować tylko chwile aktywne na magistrali OPB
- Spróbuj zarejestrować tylko chwile aktywne na magistrali OPB, kiedy procesor kontaktuje się z UARTem.

12 Software Debugger

Pakiet EDK posiada wbudowane narzędzie służące do sprzętowego debugowania procesora MicroBlaze. Aby możliwe było śledzenie pracy procesora należy wykonać szereg czynności w programie EDK:

W oknie System, klikając prawym przyciskiem myszki na mikroprocesor MicroBlaze należy ustawić S/W settings zgodnie z poniższym rysunkiem:



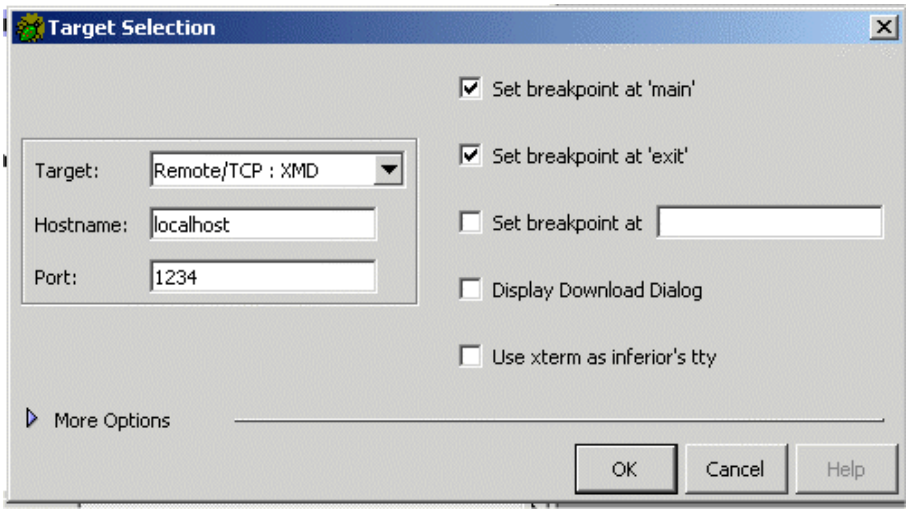
Ponadto w zakładce Optimization należy wybrać: Create Symbols for Debuging, dzięki czemu podczas kompilacji zostanie wygenerowany kod, który umożliwi debugowanie.
Następnie należy powtórnie skompilować program (menu: Tools/ Compile Program Sources) i podobnie jak to było dla pliku executable.elf wygenerować program ładowany do pamięci SRAM poprzez komendę w menu Tools/Xygwin Shell komendami:
mb-objcopy xmdstub.elf xmdstub.bin –O binary
mb-objdump -d xmdstub.elf >xmdstub.txt.

Następnie należy wybrać menu: Tools/XMD oraz uruchomić komendę:
mbconnect stub –comm serial –port com1 –baud 9600

Niestety połączenie nie zostało nawiązane poprawnie ponieważ procesor MicroBlaze nie został załadowany poprawnym kodem. Teraz to plik xmdstub.bin jest kodem, który ma wykonywać mikroprocesor i który należy załadować do pamięci SRAM w miejsce poprzedniego sramin.bin. Skrypt *apsi_csim.txt* zawiera już odpowiednie instrukcje ładujące właściwy program dlatego należy go uruchomić. Po uruchomieniu skryptu należy powtórnie uruchomić komendę *mbconnect* z takimi samymi parametrami jak poprzednio.

Analizator stanów logicznych śledzi postępowanie programu *mbconnect* i jego komunikacji z procesorem. Zobacz na ustawienia trigger'a analizatora oraz na ustawienia logiki CED. Co znajduje się pod adresem E0_0008 (zob. dokumentacje UARTa) oraz co oznacza bit 1?
latvalue E0_0008_0000_0001_02 // UART, status, received a byte and ack=1
latcare FF_FFFF_0000_0001_02
Zaobserwuj co dalej wykonuje mikroprocesor po otrzymaniu komunikacji z programem XMD. Zaobserwuj wybierając odpowiednie ustalenia logiki CED jakie dane są wysyłane i pobierane przez UART. Spróbuj sprawdzić co robi MicroBlaze po nawiązaniu łączności. Pomocny jest w tym plik *xmdstub.txt.*

Jeżeli łączność została poprawnie nawiązana należy w programie EDK wybrać menu Tools/ Software Debugger. W programie tym należy w menu File/Target Settings wybrać następujące ustawienia:



Następnie możliwe jest śledzenie pracy procesora MicroBlaze. Zaobserwuj analizatorem stanów logicznych w jaki sposób odbywa się pobieranie kodu programu executable.elf przez ten system – nie został on załadowany przecież do pamięci SRAM.